

Look inside Mac OS X

– is it as secure as we expected?

Xu Hao

windknown@hotmail.com

Agenda

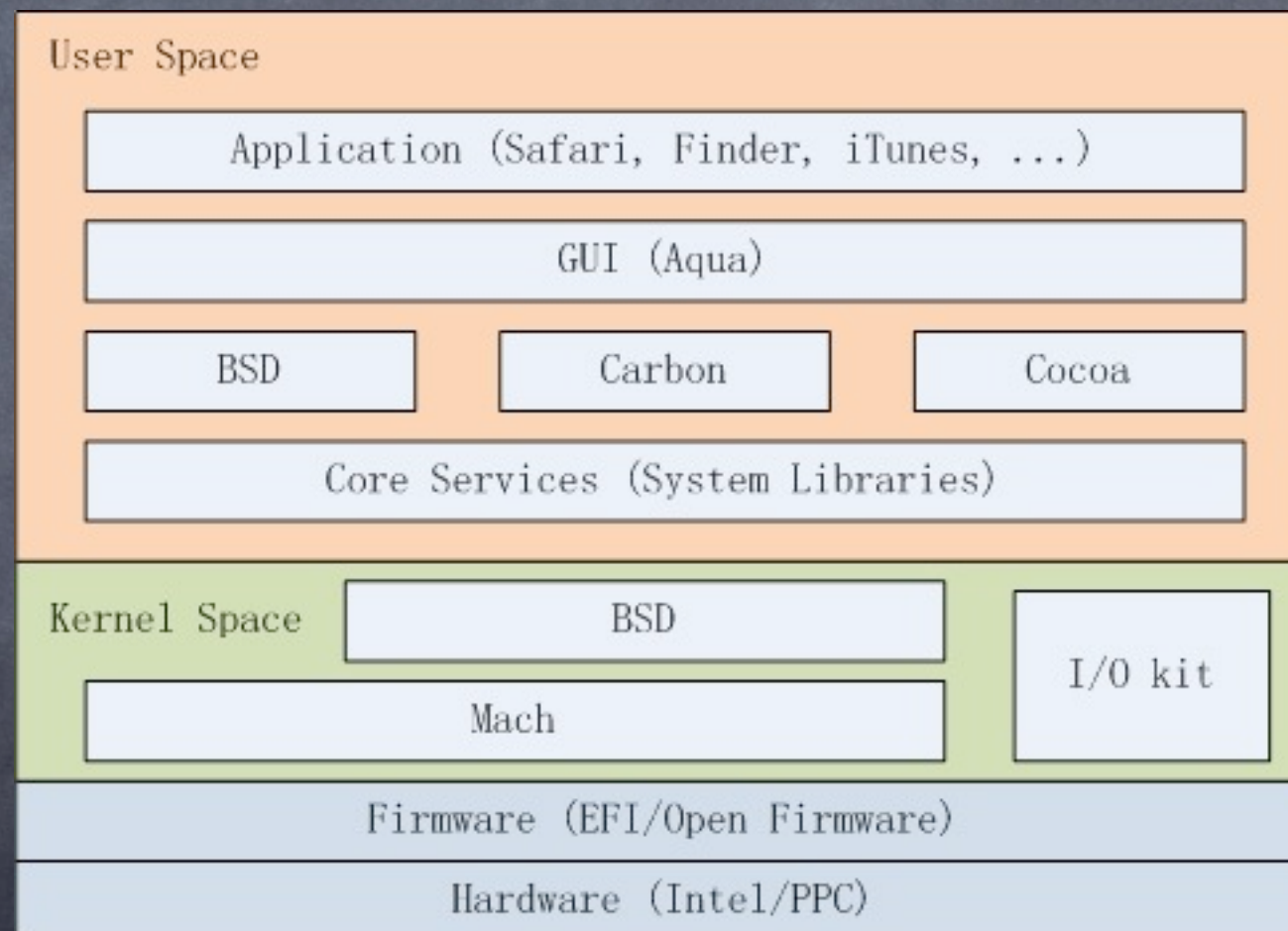
- Mac OS Introduction
- Security mechanism
- Malware
- Exploit
- Rootkit
- Conclusion

Mac OS Introduction

- Used by Apple product
 - MacBook, MacBook Pro, MacBook Air, iMac, Mac mini ...
- Mac OS 10.5.x Leopard
 - Support PowerPC / Intel, 32 bit
- Mac OS 10.6.x Snow Leopard
 - Support Intel only, 64 bit (kernel and most applications)
- Some important words
 - Darwin, XNU, BSD, Mach, Carbon, Cocoa, ...

Mac OS Introduction

• High level overview



Mac OS Introduction

- Darwin package

- A collection of open source projects include xnu kernel and a lot of tools
- <http://www.opensource.apple.com/>

- XNU kernel

- combined with Mach and BSD
- Mach
 - low level: processor, virtual memory, kernel debugging, ...
- BSD
 - high level: system calls, file system, socket, ...

Mac OS Introduction

- Program environment
 - I/O Kit
 - Develop kernel module with C++ language
- BSD environment
 - C language
 - POSIX APIs: /usr/include (open/close/execv ...)
- Carbon environment
 - C language
 - Old Mac OS style API: Carbon.h (GetProcessPID ...)
- Cocoa environment
 - Object C – superset of C language
 - Easiest way to build Mac applications, NS* API

Agenda

- Mac OS Introduction
- Security mechanism
- Malware
- Exploit
- Rootkit
- Conclusion

Security mechanism

• Privilege

- Need root privilege to control system
- Much better than old Windows: 2K/XP/2K3
- UAC is included in Vista/Win7

• Firewall

- Enabled by default in Snow Leopard
- Only can block incoming connections
- Can't set complex rules through UI
- Windows firewall is better

Security mechanism

- Library randomization
 - Libraries will be loaded according to map files
 - `/var/db/dyld/dyld_shared_cache_i386.map`
 - `/var/db/dyld/dyld_shared_cache_x86_64.map`
 - dyld load address is constant
 - The executable file address is not random
 - Third party dynamic modules address are not random
 - The map files are not changed until system update
 - ASLR in Vista/7 is better
 - all the modules address are random include exe file

Security mechanism

- Executable memory protection
 - Hardware support – Intel(XD bit)/AMD(NX bit)
 - use vmmap to verify memory protection mask
 - VM_ALLOCATE 306de000-306df000 [4K] rw-/rwx
 - Stack b0083000-b0104000 [516K] rw-/rwx
 - But truth is that stack is not executable but heap memory is executable
- DEP in Windows
 - More flexible – off/optin/optout/on
 - Also protect heap memory

Security mechanism

- Stack-Smashing Protector

- Use gcc and add `-fstack-protector`

- The beginning of function

- `mov edx, ds:____stack_chk_guard_ptr`
`mov ecx, [edx]`
`mov [ebp+var_1C], ecx`

- Before return

- `mov edx, ds:____stack_chk_guard_ptr`
`mov ecx, [ebp+var_1C]`
`xor ecx, [edx]`

- Similar with `/GS` flag in VS

Security mechanism

• Sandbox

- Restrict application access to system resource such as hard disk, network ...
- Implemented in kernel space, call `sandbox_init()` to put process in sandbox
- Use `sandbox-exec` to test, rules file examples can be found: `/usr/share/sandbox/*.sb`
- It's hard to write rules for complex applications
- No similar mechanism in Windows

Security mechanism

- `__PAGEZERO`

- The first 4K page(0-0x1000) cannot be accessed
- Segment named `__PAGEZERO` fills the first 4K page

- FileVault

- Auto transparent encryption and decryption of HOME folder
- EFS in Windows

- 32/64 bit problem

- 64 bit mode is default turned on from Snow Leopard
- But many third party softwares are not moved to 64 bit yet, such as firefox, adobe reader ...

Agenda

- Mac OS Introduction
- Security mechanism
- Malware
- Exploit
- Rootkit
- Conclusion

Malware

• Virus

- Long history and countless kinds of virus in Windows environment
- Virus using PE infection technique are dangerous and hard to clear
- Is Mac OS immune to virus?
- Executable files, dynamic modules, kernel extensions are all in Mach-O file format
- Infect Mach-O files is possible and not difficult

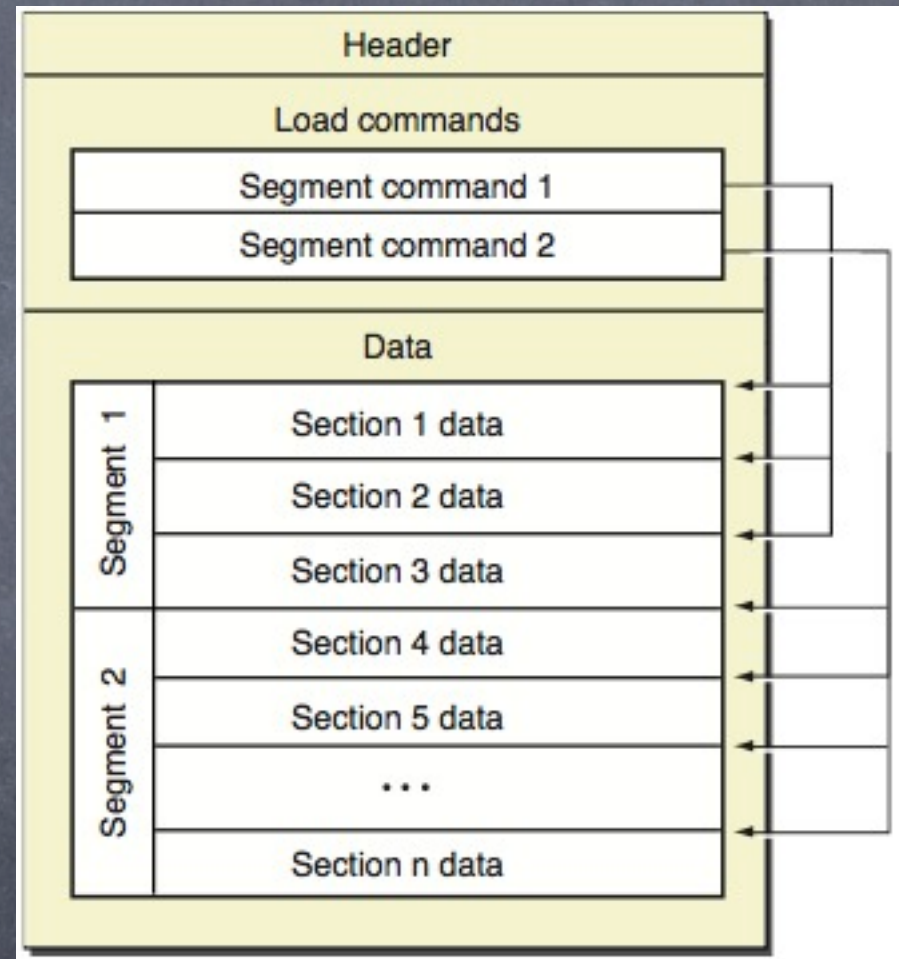
Mach-O Format

- Check first 4 bytes signature:

- 0xfeedface – 32 bit
- 0xfeedfacf – 64 bit

- Header +
Load commands +
segments
- Structure definition
can be found

- `/usr/include/mach-o/loader.h`



Mach-O Format

- Header structure
 - Specify cpu type, number of load commands and size of commands
- Load commands
 - Begin with command type and size of command
 - LC_SEGMENT - tell loader how to map the segment, include all sections
 - LC_SYMTAB - symbol table used by linkers
 - LC_LOAD_DYLINKER - set dynamic linker: /usr/lib/dyld
 - LC_UNIXTHREAD - initial thread state, include all register values, eip points to the entry point
 - LC_LOAD_DYLIB - dynamic share library need to link

Mach-O Format

- Segments
 - Named in uppercase, may contain several sections, section names are in lowercase
 - `__PAGEZERO` - protect NULL address
 - `__TEXT` - executable code and read-only data
 - `__text` - only contain executable code
 - `__cstring` - const C strings used by code
 - `__DATA` - writable data
 - `__data` - mutable variables
 - `__LINKEDIT` - some data used by linker, such as symbol names

Mach-O Format

- Use "otool -l" to see some examples

```
Load command 1
  cmd LC_SEGMENT
  cmdsize 464
  segname __TEXT
  vmaddr 0x00001000
  vmsize 0x00021000
  fileoff 0
  filesize 135168
```

```
Section
  sectname __cstring
  segname __TEXT
  addr 0x0001d1e8
  size 0x00004635
  offset 115176
  align 2^2 (4)
  reloff 0
  nreloc 0
  flags 0x00000002
```

```
Load command 9
  cmd LC_UNIXTHREAD
  cmdsize 80
  flavor i386_THREAD_STATE
  count i386_THREAD_STATE_COUNT
    eax 0x00000000 ebx 0x00000000 ecx 0x00000000 edx 0x00000000
    edi 0x00000000 esi 0x00000000 ebp 0x00000000 esp 0x00000000
    ss 0x00000000 eflags 0x00000000 eip 0x000020f4 cs 0x00000000
    ds 0x00000000 es 0x00000000 fs 0x00000000 gs 0x00000000
Load command 10
  cmd LC_LOAD_DYLIB
  cmdsize 56
  name /usr/lib/libncurses.5.4.dylib (offset 24)
  time stamp 2 Thu Jan 1 08:30:02 1970
  current version 5.4.0
  compatibility version 5.4.0
```


Malware

- Mach-O file infection
 - Expand the last segment to contain our shellcode and any other data
 - find the last LC_SEGMENT load command
 - modify vmsize/filesize/initprot
 - Change eip of LC_UNIXTHREAD command to point to shellcode
- Shellcode
 - execute whatever function, eg: fork a new process
 - jump back to raw entry point
 - shellcode details will be discussed in exploit chapter

Malware

- Universal File Format

- Contains Mach-O files for different architecture

- PPC/Intel x86/Intel x64

- /usr/include/mach-o/fat.h

- fat_header + array of fat_arch

- fat_header.magic - 0xcafebabe

- fat_header.nfat_arch - count of fat_arch

- fat_arch.offset/fat_arch.size - Mach-O file body

- Infection

- Locate Mach-O file body and infect

- Note that the fat_header/fat_arch are stored in big endian

- Modify fat_arch.offset and fat_arch.size

Malware

- Demo – Infect ls
 - Expand ls file – increase 64KB
 - Shellcode will print some information before normal function to be executed

```
windknownmatoMacBook-Pro:bin windknown$ ./ls -l
total 96
-rwxr-xr-x  1 windknown  staff   9840  7  6 15:15  dinfect
drwxr-xr-x  3 windknown  staff    102  7  6 15:15  dinfect.dSYM
-rwxr-xr-x  1 windknown  staff  35632  7  6 15:43  ls
windknownmatoMacBook-Pro:bin windknown$ ./dinfect ls
[OK] last segment: __LINKEDIT vmaddr: 7000 vmsize: 3000 fileoff: 24576 filesize: 11056
[OK] original eip: 1a60
windknownmatoMacBook-Pro:bin windknown$ ./ls -l
+++++++This file is already infected+++++++
total 224
-rwxr-xr-x  1 windknown  staff   9840  7  6 15:15  dinfect
drwxr-xr-x  3 windknown  staff    102  7  6 15:15  dinfect.dSYM
-rwxr-xr-x  1 windknown  staff 101168  7  6 15:43  ls
```


Malware

- Trojan horse

- Network connection

- Windows – need to bypass firewall
 - Mac OS – outcome connections are always allowed

- Shell

- Windows – cmd.exe
 - Mac OS – /bin/bash

- Auto start

- Windows – modify system registry
 - Mac OS – plist file

Mac OS Startup

- Kernel-level init - /mach_kernel
 - Basic kernel components - BSD, Mach, IOKit
 - /System/Library/Extensions/System.kext/PlugIns/*.kext
 - Other kernel extensions
 - /System/Library/Extensions/*.kext
 - Info.plist
 - OSBundleRequired - Root/Local-Root/Network-Root/Safe Boot
- User-level init - /sbin/launchd
 - Load agents and daemons according to plist files
 - /System/Library/LaunchAgents /System/Library/LaunchDaemons
 - /Library/LaunchAgents /Library/LaunchDaemons
 - ~/Library/LaunchAgents

Mac OS Startup

- launchctl load/unload/list
- plist file example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.sourceforge.macgpg2.gpg-agent</string>
  <key>RunAtLoad</key>
  <true/>
  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/gpg-agent</string>
    <string>--launchd</string>
    <string>--write-env-file</string>
    <string>--use-standard-socket</string>
  </array>
  <key>LimitLoadToSessionType</key>
  <string>Background</string>
  <key>ServiceIPC</key>
  <true/>
</dict>
</plist>
```


Malware

- Keylogger implement
 - similar to message hook technique in Windows
 - filter key press event
 - CGEventTapCreate to create an event tap
 - eventsOfInterest - kCGEventKeyDown/kCGEventKeyUp
 - callback - our filter function
 - CFMachPortCreateRunLoopSource to create a run loop event source
 - CFRunLoopAddSource to add the source to run loop
 - Access for assistive device must be enabled

Agenda

- Mac OS Introduction
- Security mechanism
- Malware
- Exploit
- Rootkit
- Conclusion

Exploit

- Shellcode

- Depends on CPU architecture – PPC/Intel

- Apple only supports Intel CPU from Snow Leopard

- Depends on OS

- We are familiar with shellcode for Windows

- Shellcode for Mac OS 32 bit and 64 bit are little difference

- What we discussed today

- User-mode shellcode and kernel-mode shellcode

- Environment – Intel CPU and Mac OS 32 bit

User-mode Shellcode

- Shellcode functions
 - execute file
 - download and execute file
 - reverse socket shell
 - ...
- Keypoint – call APIs
 - Directly call system API
 - Locate useful API address

User-mode Shellcode

- Call system API
 - User-mode API → system API provided by OS kernel
 - Windows
 - int 2eh / sysenter
 - Mac OS
 - int 80h / sysenter
 - Mac OS syscall
 - eax specify which API to be called
 - call number can be found: `/usr/include/sys/syscall.h`

User-mode Shellcode

- fork() example – #define SYS_fork 2

```
_fork      proc near  
  
dyld_func_name = dword ptr -1Ch  
address       = dword ptr -18h  
var_14        = dword ptr -14h  
var_10        = dword ptr -10h  
  
          sub     esp, 1Ch      ; __fork  
          call    $+5  
  
loc_2C8AC:                                ; DATA XREF: _fork+73↓w  
          pop     edx  
          mov     edx, ds:(__cthread_fork_prepare_ptr - 2C8ACh)[edx]  
          call    edx  
          mov     eax, 2  
          int     80h
```

- getppid() example – #define SYS_getppid 39

```
_getppid   proc near  
          mov     eax, 27h ; '' ; __getppid  
          call    __sysenter_trap  
          jnb     short locret_7671E
```

```
__sysenter_trap proc near  
  
          pop     edx  
          mov     ecx, esp  
          sysenter
```


User-mode Shellcode

- Locate API address in Mach-O file format
 - LC_SYMTAB command
 - symoff – the location of symbol table entries
 - stroff – the location of string table
 - Symbol table is an array of nlist structure
 - n_un.n_strx – index in string table
 - n_value – address of the symbol
- Compare symbol string by hash value

User-mode Shellcode

• Method A

- dyld base address - 0x8fe00000
- Locate "__dyld_get_all_image_infos" function
- Get pointer to "dyld_all_image_infos" struct
 - dyld_all_image_infos.infoArray points to "dyld_image_info" struct
 - dyld_image_info.imageLoadAddress / dyld_image_info.imageFilePath
- Find module base address by name and locate API address

• Method B

- dyld base address - 0x8fe00000
- Locate "_dlopen" and "_dlsym" function
- Find API address we need

Real Exploit

- Adobe Reader Exploit Demo
 - Fill memory with shellcode
 - js heap spray
 - Trigger vulnerability
 - jump to shellcode
 - Shellcode function
 - fork a new process and execute TextEdit

Comparison

- Windows 2K/XP/2K3
 - DEP – heap memory are none-executable
 - Return-oriented programming
- Windows Vista/2K8/7
 - ASLR – can't find constant instruction address to ret
 - JIT spray to allocate executable memory
 - Exploit a memory disclosure vulnerability to determine at least one dll base address
- Mac OS 10.5/10.6
 - Heap is executable
 - dyld base address is constant

Kernel-mode Shellcode

- Destination

- Gain root privilege for normal user process

- Environment

- Shellcode is executed in kernel mode

- Method

- Change process user id to root (0)
 - Must keep kernel running

Kernel-mode Shellcode

- Get proc struct address of current process
 - call sysctl to get kinfo_proc struct

```
int mib[4];
struct kinfo_proc my_proc;
size_t len;

mib[0] = CTL_KERN;
mib[1] = KERN_PROC;
mib[2] = KERN_PROC_PID;
mib[3] = getpid();
len = sizeof(my_proc);
sysctl(mib, 4, &my_proc, &len, NULL, 0);
```

- kinfo_proc.kp_eproc.e_paddr
 - point to proc struct address

Kernel-mode Shellcode

- Find ucred pointer in proc struct

- Locate "ucred *proc_ucred(proc *p)"

```
_proc_ucred:
0047cdda      pushl    %ebp
0047cddb      movl     %esp,%ebp
0047cddd      movl     0x08(%ebp),%eax
0047cde0      movl     0x64(%eax),%eax
0047cde3      leave
0047cde4      ret
```

- Set user id of ucred to zero

- Get cr_uid/cr_ruid offset in ucred struct

- Locate "uid_t crgetuid(ucred *p)"

```
_crgetuid:
004e691f      pushl    %ebp
004e6920      movl     %esp,%ebp
004e6922      movl     0x08(%ebp),%eax
004e6925      movl     0x0c(%eax),%eax
004e6928      leave
004e6929      ret
```


Agenda

- Mac OS Introduction
- Security mechanism
- Malware
- Exploit
- Rootkit
- Conclusion

Rootkit

- Modify OS kernel to hide something, such as file, process, module, connection
 - Rootkit code must be loaded into kernel space
- Mac OS kernel module development
 - Normal kernel Extension - Written in C
 - I/O kit framework - Written in C++
- Compared with rootkit of Windows
 - Similar ideas
 - different implementation
- We will only discuss some basic ideas here to prove that it is not so hard to develop a Mac OS rootkit

Rootkit

- Hook system call
 - Just like hook SSDT of Windows kernel
 - Mac system call table
 - definitions can be found - `bsd/sys/Sysent.h`
 - `extern struct sysent sysent[];`
 - Array of sysent struct to hold all system call
 - `extern int nsysent;`
 - Max system function count
 - `NUM_SYSENT / SYS_MAXSYSCALL`
 - To hook system call - need to find sysent address

Rootkit

- Locate sysent table address
 - sysent symbol is not exported
 - but nsysent symbol is exported
 - Actually nsysent is just behind sysent table
- Find nsysent and exit function address
 - Reverse search 4-bytes value match exit function address
 - Since call number of exit function is 1 (the second one in sysent table), sysent address can be determined
 - sysent.sy_call points to service function address

Rootkit

- Steps to hide something
 - Modify sysent to hook certain function
 - Call original function
 - Modify output data to be returned to user
 - system call definition
 - `int32_t sy_call_t(struct proc *, void *, int *);`
 - the 2nd parameter points to function arguments list address, the 3rd parameter will store the return value
 - `/System/Library/Frameworks/Kernel.framework/Versions/Current/Headers/sys/sysproto.h`
 - Use `copyin()` to read user space memory and modify data according to certain demand
 - Use `copyout()` to write data back to user space memory

Rootkit

- Some examples

- Hide files

- `getdirentries/getdirentries64/getdirentriesattr`

- Hide network connections

- `sysctlbyname` - "net.inet.tcp.pcblist"

- first call `sysctl` with {0, 3} to translate name into mib

- second call `sysctl` with mib value

- Hide process

- `sysctl` - {CTL_KERN, KERN_PROC, KERN_PROC_ALL}

Rootkit

- Communicate with user space process
 - Unlike Windows, Mac OS kernel memory is not mapped into user process space
 - Dynamically add our own sysctl name
 - SYSCTL_PROC macro to create a sysctl_oid struct
 - specify handler/type/args/name/oid ...
 - int oid_handler SYSCTL_HANDLER_ARGS {}
- Register handle function
 - sysctl_register_oid() to register the sysctl_oid struct we created before
- User process call sysctl

Rootkit

- Anything else to hook? (1)
 - Just like rootkit under Windows, different kinds of techniques can be used to modify kernel
 - DKOM
 - Hide process – extern struc proclist allpro;
 - Hide kernel module – extern kmod_info *kmod;
 - Other methods to hook system call
 - How about inline hook? No patch guard!
 - int 80h – hi_unix_call – lo_unix_call – unix_syscall
 - sysenter – hi_sysenter – lo_sysenter – unix_syscall

Rootkit

- Anything else to hook? (2)
 - sysctl_oid hook
 - sysctl() will find certain sysctl_oid struct according to mib value and call the handler function
 - We can modify the handler function address stored in sysctl_oid struct
 - Search sysctl_oid struct
 - By name
 - "net.inet.tcp.pcblist" - `_sysctl__net_inet_tcp_pcblist`
 - By oid value
 - search from root struct - `_sysctl_hw/_sysctl_kern/...`

Agenda

- Mac OS Introduction
- Security mechanism
- Malware
- Exploit
- Rootkit
- Conclusion

Conclusion

- Nearly all attack vectors under Windows can be shifted to Mac OS platform
- On some aspects, Windows security mechanism is better
- Mac OS seems to be safer for lack of attackers – Windows are explored by thousands of researchers for many years
- Open source strategy has both advantage and disadvantage
- Mac OS is really cool and interesting, not only for normal users but also for security researcher. There are still many things to be explored, such as rootkit, bootkit, VM rootkit, EFI security, etc.

Any Question?

Thanks for ur time & hope u have fun

windknown@hotmail.com